Dataflow/Actor-Oriented language for the design of complex signal processing systems

Christophe Lucarz*, Marco Mattavelli*,
Matthieu Wipliez†, Ghislain Roquier†, Mickaël Raulet†,
Jörn W. Janneck‡, Ian D. Miller‡ and David B. Parlour‡
*Ecole Polytechnique Fédérale de Lausanne (Switzerland)
†IETR Laboratory - UMR CNRS 6164 - Rennes (France)
‡Xilinx Inc. - San Jose (U.S.A)

***Abstract*— Signal processing algorithms become more and more complex and the algorithm architecture adaptation and design processes cannot any longer rely only on the intuition of the designers to build efficient systems. Specific tools and methods are needed to cope with the increasing complexity of both algorithms and platforms. This paper presents a new framework which allows the specification, design, simulation and implementation of a system operating at a higher level of abstraction compared to current approaches. The framework is base on the usage of a new actor/dataflow oriented language called CAL. Such language has been specifically designed for modelling complex signal processing systems. CAL data flow models expose the intrinsic concurrency of the algorithms by employing the notions of actor programming and dataflow. Concurrency and parallelism are very important aspects of embedded system design as we enter in the multicore era. The design framework is composed by a simulation platform and by Cal2C and CAL2HDL code generators. This paper described in details the principles on which such code generators are based and shows how efficient software (C) and hardware (VHDL and Verilog) code can be generated by appropriate CAL models. Results on a real design case, a MPEG-4 Simple Profile decoder, show that systems obtained with the hardware code generator outperform the hand written VHDL version both in terms of performance and resource usage. Concerning the C code generator results, the results show that the synthesized C-software mapped on a SystemC scheduler platform, is much faster than the simulated CAL dataflow program and approaches handwritten C versions.**

I. Introduction

With the unbounded increase of signal processing systems complexity made possible by both the advances in algorithm theory and by new generations of silicon technology, digital systems designers need new tools for the design of efficient systems employing "reasonable" design resources. Increasing the level of abstraction has always been a solution to appropriately handle the increasing complexity of systems design. Transistors, gates, VHDL and Intellectual Property (IP) blocks are example of different abstractions layers which have been successfully introduced in the past with the attempt of easing the design of more and more complex systems. In the authors opinion the current new challenge is not only to add a new abstraction layer, but also to close the gap between the specification and the implementation layers. Such gap, in the recent years, has been tried to be filled by using C/C++ reference implementations of the specifications. However, the path from such specifications expressed as generic C/C++ reference SW to gate design has shown to be harder to be efficiently accomplished. One of the main reasons of such difficulty is the fact that C/C++ do not provide the operators that enable to naturally express parallelism, data flows and other fundamental architectural features without adding a huge amount of low level programming details. In other words the operators are defined at a level that is by, far too, low compared to the one at which a designer would like to express his architectural ideas. Also other approaches and methods aiming at closing the gap from specification to synthesis have yet to deliver on their promise. Several methodologies based on using UML, different variants of C/C++ (i.e streamC [1]) combined with SystemC/TLM library at different levels of the design flow are used. Some of them make use of exploration tools, simulators and code generators which can output hardware and software code specific to FPGA or other processor platforms, mainly using VHDL and C. Section II briefly reviews these different approaches and methods for building digital systems.

This paper presents a new design framework, currently in its early phase of development, based on CAL language, an actor and dataflow oriented language designed for the specification of complex signal processing systems. The new framework includes: a simulator of the system specified by CAL, a hardware generator for the direct synthesis of HDL and a software generator for the synthesis of C from CAL. The paper explains why and how this framework is a very promising approach for the design and development complex heterogeneous processing systems.

Section 1 introduces the CAL language, a dataflow and actor-based language that constitutes a very interesting candidate to support a design flow for designing complex signal processing systems. The major features of the new approach are simplicity, conciseness and expressiveness. Modelling systems in a concise and simple way is a good starting point, but at the same time enabling the automatic generation of efficient code is an extremely challenging step. Sections IV and V show how efficient hardware and software code can be generated directly from CAL language system descriptions.

Section VI discusses the reasons for which CAL language and the associated framework are a good approach to support a design flow for building complex heterogeneous systems.

Section VII concludes the paper.

II. Related Work

This section provides a brief overview of the current design flows based on UML, SystemC, and C/C++. A comprehensive overview of the state of the art of system level modeling can be found in [2].

Although UML was originally conceived for modelling large software systems, Kukkala et al. [3] developed a design flow for multiprocessors Systems on Chip (SoC) by using UML 2.0 models as the starting point of the design flow. UML models (i.e. application, platform and mapping models) are written according to the experience of the designers. However, there is limited support for the elaboration of UML specifications. When building a system from monolithic C/C++ specifications (such as one of the the MPEG reference softwares for instance) UML models must be completely written by hand. This task is very time-consuming and error prone. Furthermore, according to [4], "*UML 2.0 lacks both a reference implementation and a human-readable semantic*".

Modelling HW using a high level description languages such as SystemC can be a solution for representing functionality, concurrency, communication, software and hardware components at different (system) levels of abstraction. A language modelling complex systems should also provide a support for analysis and synthesis. SystemC is mainly used for simulation because it is not synthesisable in its whole generality. In practice, it can be reduced for ensuring the use of a synthesizable subset. The problem is that this subset of the language is placed at a quite low level of abstraction. Simulation capabilities at a high level of abstraction are an interesting feature of SystemC, but during the implementation step, the designer is forced to re-write the code using only the synthesisable subset. Such operation is a time consuming and error-prone task. In a nutshell, it is hard to use SystemC as a high level design language because in general it is not possible to implement directly systems just developing at high level the system specification.

Several tools propose a direct conversion of C code into VHDL. C-based methodologies for modelling systems lack concurrency and a concept of time. Hardware is inherently parallel and time is essential to represent its behaviour accurately, especially for real-time embedded systems. The main problems of using C as a Hardware Description Language are discussed in detail in [5] and [6]. In conclusion, the lack of intrinsic concurrency and the concept of time as well as the way communication mechanisms are written in imperative languages are the main limitations and drawbacks of C or similar languages for hardware representation.

III. CAL a language for algorithm and architecture specification

The more important features that system designers would like to find in a design language would be the capability of representing both algorithm and architecture at a single level of abstraction. Unfortunately, it is difficult to specify an algorithm at a high level of abstraction and at the same time being able to deduce from such description an efficient low level representation. How can high level and low level constructs be combined into a single language? Furthermore, such language should support full simulation of the system behaviour. Thus, designers have been forced to use several languages to represent complex algorithms at different levels of abstraction during the entire design flow. Generic C, architectural C, SystemC, synthesizable SystemC is a possible first stage of a design methodology. The main drawback of such multistage approaches is how to implement the conversions between all these languages avoiding resource consuming hand re-writing [4]. Compilers often support only a subset of the languages, making the different conversions between the languages a headache! Thus, the reduction of the number of languages and abstraction levels composing a design flow is a fundamental issue. Ideally, the unique language transformation should be a direct translation from the high level language down to an implementation language such as C, VHDL or Verilog.

Abstract languages do not often support simulation capabilities and architectural representations are available at too low levels of abstraction. Standard imperative languages do not express easily parallelism. These facts reduce considerably the set of possible candidates. The new actor/dataflow oriented language called CAL has been specifically developed so as to support the features discussed above.

A. CAL language

CAL is a dataflow and actor oriented language that has been recently specified and developed by one of the authors of this paper as a subproject of the Ptolemy project at the University of California at Berkeley. The final CAL language specification has been released in December 2003 [7]. CAL describes algorithms by using a set of encapsulated dataflow components called *actors* communicating with each others. An actor is a modular component that encapsulates its own state. The state of any actor is not shareable with other actors. Thus, an actor cannot modify the state of another actor. The only interaction an actor has with another actor is only thought inputs and outputs ports. The behaviour of an actor is defined in terms of a set of *actions*. The operations an action can perform are to consume (read) input tokens, to modify internal state, to produce output tokens. The topology of the connections between actors input and output ports constitute what is called a network of actors. It is expressed by using an XML dialect known as network language (NL) that also includes the possibility to includes attributes (i.e. parameters) that may be different for the instantiation of the same (parametric) actor in a network of actors. Each action of an actor defines the kind of transitions that internal states can undergo and actions can be fired under specific conditions: (1) the availability of input tokens, (2) the value of input tokens, (3) the state of the actor or (4) the priority of that action. In an actor, the operations are executed sequentially. The execution of actions follows the following cycle:

1) Determine, for each action, whether it is enabled, by

testing all the conditions specified in that action. It depends on the availability of token(s) at the requisite input(s), the value of input tokens, the state of the actor and the priority of each action.
2) If one or more actions are enabled, pick one of them to be fired next;
3) Fire that action.
4) Go to (1).

The transitions of an actor are purely sequential: actions are fired one after another. At the network level, the actors are completely independent and can work concurrently, each one executing their own sequential operations. Figure 1 illustrates a CAL model in general and the reader can find examples of CAL actors in figures 4 and 5(a).

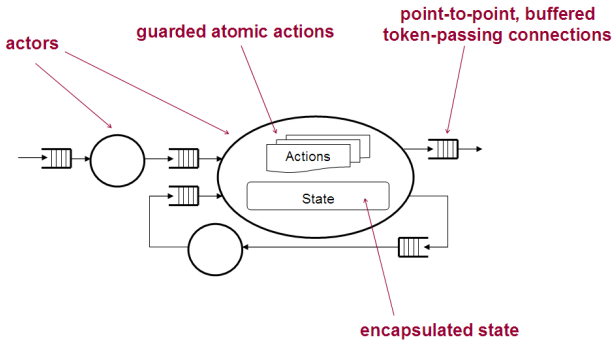

Fig. 1. Illustration of a CAL model

The selection order and the firing conditions for actions form the core of the design of an actor. CAL provides a number of constructs for describing action selection, which include *guards* (conditions on the values of input tokens and/or the values of actor state variables), a *Finite State Machine* and *priorities* (actions may be related to each other by a partial priority order). The order of execution of the actors is in general not known *a priori*. CAL provides a great flexibility to schedule action execution according to the particular requirements and constraints of the targeted device chosen for the final implementation.

CAL language very naturally allows also hierarchical system design. Each actor can be specified as a network of actors. This approach facilitates modularity, where the internal specification of any actor can be modified without impacting other actors.

B. Goals and features of CAL language

Ease of use CAL is a true programming language and not an intermediate format to automatically generate code. The notation is convenient to write, with consistent syntax rules, keywords, and structures.

Minimal semantic core CAL language is built on a very small set of semantic concepts. It simplifies compiler construction when transforming any program into an equivalent program in the core language. Thus for the compilation to any given platform, a code generator is needed for the specific implementation language.

Implementation independence and retargetability The first target for CAL actors was the Ptolemy II platform [8]. A complete framework (Open DataFlow) is currently being developed for simulating CAL networks and for generating hardware and software code [9]. OpenDF is built under the Eclipse environment. It makes these tools available for a large set of operating systems. CAL models are technology and architecture agnostic. Thus, it makes possible to design and simulate models very quickly using [9]. Designers do not need special hardware or libraries to design their own system in CAL. The implementation of CAL models is done by means of appropriate hardware and software code generators described in more details in the following sections..

C. Framework infrastructure

Figure 2 presents the general framework infrastructure. Once the CAL model is defined, the user can simulate it using the OpenDF simulator [9]. The user can generate software code and hardware code from the CAL model. These code generators are detailed in section IV and V respectively.

IV. Software Synthesis

A. Semantics of CAL dataflow

The system behavior of a dataflow program is determined by the interactions between actors (i.e. exchange of data tokens). Such interactions are governed by a Model of Computation (MoC) that defines which scheduling policies can be used to fire actors. The CAL language is not related to any particular dataflow MoCs. Indeed, several forms of dataflow exist to interpret the network from the general dataflow process network (PN) [10] model with multiple firing rules to the more restrictive synchronous dataflow (SDF) one [11], [12]. CAL extends the model in [10] by :

1) state within actors,
2) multiple overlapping (non-joinable in the parlance of [10]) firing rules, and
3) priorities among firing rules.

CAL actors often contain multiple actions and priorities, FSM or guards that lead to state-dependent or conditional execution. Therefore most of the CAL actors in the library are closer to the PN model which is then chosen (as a first milestone) for developing the tool described in this paper.

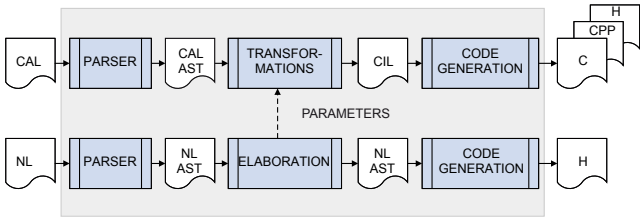

Fig. 3. Cal2C compilation process

When the PN model is chosen to interpret CAL networks, any environment which supports multithreading may be chosen for implementation. For instance, it can be done

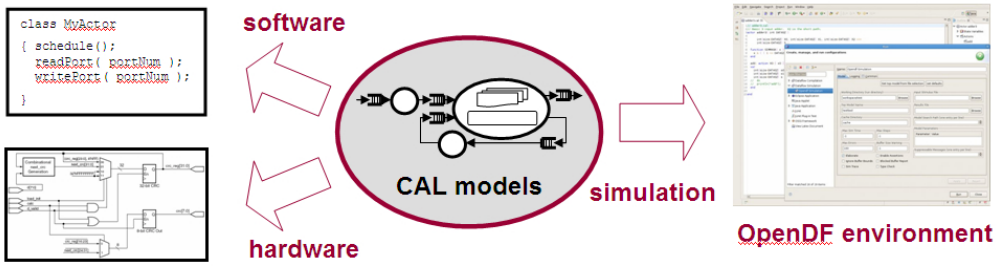

Fig. 2. The framework infrastructure

using POSIX threads by translating CAL actors into threads and by replacing connections with FIFOs. However, it is necessary to develop a scheduler and low-level considerations such as communication or actual scheduler implementation, make this solution time-consuming and error prone. Another approach for PN implementation is provided by the SystemC[1] standard whose simulation environment permits high-level programming, well-adapted to functional verifications. A PN-oriented SystemC model is expressed as a network of modules communicating with each other via blocking FIFOs (with an additional way to support token availability). Note that Cal2C does not intend to use SystemC as an hardware description language, but only as a convenient PN modeling and simulation environment. Moreover, this work is a first step to an efficient "pure C" code generation (closer to an SDF model than a PN one). In short, the software synthesis from a network of CAL actors produces several files as illustrated in Figure 3. Each NL network is translated into a header file where FIFOs, modules and sub-networks are instantiated and connected. CAL actor translation is done in 2 different parts: the actor code (actions, functions, procedures) to express the functionality and the action scheduler (priorities, FSMs, guards) to control the execution. Finally an additional file is created to instantiate the top network and to launch SystemC simulation.

B. C code generation: Actors

Actor code translation: Translating the CAL actor code produces a single C file wherein functions, procedures and actions are translated. however, C language and compilers impose limitations on the translated code. For example, (1) distinct functions should have different names to avoid linking problems, even if they are originally placed in different actors. Another challenge is (2) the difference of programming paradigm between the source and the target language: CAL allows functional constructs that have no direct equivalent in C. The action translation process starts with an Abstract Syntax Tree (AST) issued from the CAL source code, and modifies it as needed to satisfy the previous requirements. Function names are prefixed with the actor name to prevent any potential name clashes; actor parameters are replaced by their values (when constant) or transformed to local variables otherwise; actions are converted to functions where input and

[1]http://www.systemc.org

```
ac: action AC:[i] ⇒ OUT:[ saturate( o )]
var
  int(size=SAMPLE_SZ) v =
    ( quant * ( lshift( abs(i), 1) + 1) ) - round,
  int(size=SAMPLE_SZ) o =
    if i = 0 then 0 else if i < 0 then -v else v end end
do
  count := count + 1;
end
```

(a) CAL "ac" action in the "Inversequant" actor

```
void Inversequant_ac(
        struct Inversequant_variables *_actor_variables ,
        int i , int *out )
{ int v, o ;
  int _call_6, _call_9;
  int _if_7, _if_8 ;
  _call_6 = Inversequant_fun_abs(_actor_variables, i);
  v = _actor_variables->quant * ((_call_6 << 1) + 1)
      - _actor_variables->round;
  if (i == 0) {
    _if_8 = 0;
  } else {
    if (i < 0) {
      _if_7 = - v;
    } else {
      _if_7 = v;
    }
    _if_8 = _if_7;
  }
  o = _if_8;
  (_actor_variables->count) ++;
  _call_9 = Inversequant_fun_saturate(_actor_variables, o);
  *out = _call_9;
}
```

(b) C translation of the "ac" action in the "Inversequant" actor

Fig. 4. C translation of a CAL action

output patterns become parameters. Finally, actor declarations are ordered by dependencies between locals, so that a variable or a function is defined before being used. At this point, the Cal2C generator converts the AST to λ-calculus, applies Damas-Milner $\mathcal{W}$-algorithm [13] to it, and augments the AST with the type information returned. Types are necessary for correct C code generation, and type-dependent transformations. Functions that return lists are "in-lined", and list sizes are computed. The transformed CAL AST is expressed in the C Intermediate Language (CIL) [14], where CAL functional constructs are replaced by imperative ones. C code is generated by calling the pretty-printer module included in the framework.

An example of the CAL actor code translation process is illustrated on figure 4. Figure 4(a) reports an "action"

extracted from the *Inversequant* actor of the RVC reference MPEG4-SP decoder, and figure 4(b) reports the corresponding generated C code. The resulting code presents a function whose name is composed of the actor name and the action name (requirement (1)). Its parameters are the same of the action's, with an additional pointer to a structure containing the actor variables. The **if** expression has been transformed to assignments of temporary variables (**_if_7**, **_if_8**) (requirement (2)). As a matter of fact, function calls have also become assignments of temporary variables (**_call_6**, **_call_9**) because CIL semantics requires it. The action output expression is translated as a pointer parameter whose value is written at the end of the C function. The synthesized C code shown in figure 4(b) is quite close to the original CAL code and results also reasonably readable by "humans".

Action scheduling: An action scheduler is created to control the action selection during execution. Priorities, guards, token consumption rates and FSM have to be translated to this end. Determining the overall order of action execution is required to have a consistent evaluation of actions that can be fired. Priorities are resolved by sorting actions in a total order and by adding a *if-then-else* statement around actions wherein the condition is given by the availability of input tokens and the guards conditions. FSMs are resolved using *switch-case* statement. Finally, the generated file consists of a thread with an infinite loop wherein its body consists of the result of the previous transformations and actions are replaced with their corresponding C functions.

For instance, a downsampler by N is illustrated figure 5(a). It could be written in different and also simpler forms, but in this form the actor enables to highlight key features of the action scheduling translation. The synthesized C++ code is illustrated in figure 5(b).

```
actor downsampler (N) In ⇒ Out :
  count := 1;
  pass: action In: [x] ⇒ Out: [x] end
  done: action ⇒ guard count = N do count := 1; end
  skip: action In: [x] ⇒ do count := count + 1; end

  schedule fsm pass:
    copy    ( pass ) --> discard;
    discard ( done ) --> copy;
    discard ( skip ) --> discard;
  end
  priority
    done > skip;
  end
end
```

(a) CAL downsampler

```
struct downsampler_vars {
   int count;
   int N;
};
void downsampler::process() {
  int fsm_state, _call_6, _call_7,_out_1;
  struct downsamplerN_vars _actor_vars;
  _actor_variables.count = 1;
  fsm_state = 1;
  while (1) {
    switch (fsm_state) {
    case 1:
      _call_6 = In->get();
      downsampler_pass(&_actor_vars, _call_6, &_out_1);
      Out->put(_out_1);
      fsm_state = 2;
      break;
    case 2:
      if (_actor_vars.count == _actor_vars.N) {
        downsampler_done(&_actor_vars);
        fsm_state = 1;
      } else {
        _call_7 = In->get();
        downsampler_skip(&_actor_vars, _call_7);
        fsm_state = 2;
      }
      break;
    }
  }
}
```

(b) Synthesized action scheduler

Fig. 5. Action scheduling (FSM) of CAL actor

C. SystemC code generation: Networks

Expressing a network of actors specified by NL in SystemC is relatively straightforward: actor or network instantiations are transformed into module instantiations. There are two semantic differences however: FIFO channels, while implicit in NL, must be explicitly created in SystemC. Broadcast of data from a source to several sinks is transparent in NL, but requires additional logic in SystemC, namely a generic broadcast module.

D. Generation of C code from a CAL description: IDCT and MPEG-4 Simple Profile decoder

So as to validate the correctness of the Cal2C code generation, the first case study is a two-dimensional inverse DCT. The IDCT is a component of all MPEG standard video decoders and is fully specified by the new Finite Precision IDCT Specification [15] based on [16]. The algorithm consists of applying one-dimensional IDCT along the row and column axis of an 8×8 pixel block. The network is composed of 2 input ports, 1 output port and 5 different actors; one actor can be instantiated several times in NL. Incoming tokens from *IN* port are inverse-quantized coefficint and a token from *SIGNED* enables to specify to the "clip" actor if incoming coefficient are signed or not. The first row of Table I shows the number of files of the respective programs. In the code generation process, an actor first is converted into a C file, then into a C++ and finally into a header file. A network of actors simply is translated into a header file while the corresponding C++ file becomes the "main" file. The second row exhibits the corresponding source lines of code (SLOC). The testbed consists of applying a stimulus (streamed by a C-code reference software of MPEG-4 SP decoder) to the top network and verifying the response against an expected result (from the CAL description simulated compared to a "golden reference" streamed by the C-code reference software).

Another synthesized model of a more complex CAL dataflow network simulated with the Open Dataflow environment has also been used to validate the Cal2C tool. The compilation process has been successfully applied to the full MPEG-4 Simple Profile dataflow model written by the MPEG

IDCT	CAL	NL	C	C++	H
Number of files	5	1	5	6	6
Code Size (SLOC)	131	25	324	386	107

TABLE I

CODE SIZE AND NUMBER OF FILES AUTOMATICALLY GENERATED FOR THE IDCT

RVC working group. Table II shows that the synthesized C-software is faster than the simulated CAL dataflow program (20 frames/s instead of 0.15 frames/s), and close to real-time for a QCIF format (25 frames/s) on a standard PC platform. It is interesting to note that the model is scalable: the number of macro-blocks decoded per second remains constant when dealing with larger image sizes.

MPEG4 SP decoder	Speed kMB/S	Code size kSLOC
CAL simulator	0.015	3.4
Cal2C	2	10.4

TABLE II

MPEG4SP DECODER SPEED AND SLOC

The MPEG4 SP dataflow program is composed of 27 actors. An actor composing a network of actors can be instantiated several times. For instance there are 42 actor instantiations in the MPEG-4 SP dataflow model. The number of SLOC generated is shown in Table III. All of the generated files are successfully compiled by gcc. For instance, the "Parser-Header" actor inside the "Parser" network is the most complex actor with multiple actions. The translated C-file (with actions and state variables) includes 1043 SLOC for actions and 1895 for action scheduling. The original CAL file contains only 962 lines of code as a comparison.

MPEG4 SP decoder	CAL	NL	C	C++	H
Number of files	27	9	27	28	36
Code Size (kSLOC)	2.9	0.5	5.8	3,7	0.9

TABLE III

CODE SIZE AND NUMBER OF FILES AUTOMATICALLY GENERATED FOR MPEG4 SP DECODER

V. HARDWARE SYNTHESIS FROM CAL DATAFLOW MODELS

In the current version of the HDL generator, each actor is translated separately into HDL and is connected with FIFO buffers in the resulting RTL descriptions. Consequently, no cross-actor optimizations are employed at the current level of development of the tool.

If two actors connected in this manner are specified to belong to different clock domains, an asynchronous FIFO implementation is selected, otherwise a synchronous FIFO is used for compactness of the implementation. Actors interact with FIFOs using a handshake protocol, which allows them to detect when a data token is available or when a FIFO is full.

The translation of each CAL actor into a hardware description follows a three-step process: (a) Instantiation, (b) Precompilation and (c) RTL code generation.

A. Instantiation

The elaboration of the network structure yields a number of actor *instances*, which are references to CAL actor descriptions along with actual values for its formal parameters. From this, instantiation computes a *closed* actor description, i.e. one without parameters, by moving the parameters along with the corresponding actual values into the actor as local (constant) declarations. It then performs constant propagation on the result.

B. Precompilation

After some simple actor *canonicalization*, in which several features of the language are translated into simpler forms, precompilation performs some basic source code transformations to make the actor more amenable to hardware implementation, such as e.g. inlining procedures and function calls. Then the canonical, closed actors are translated into a collection of communicating threads.

In the current implementation, an actor with N actions is translated into $N + 1$ threads, one for each action and another one for the *action scheduler*. The action scheduler is the mechanism that determines which action to fire next, based on the availability of tokens, the guard expression of each action (if present), the finite state machine schedule, and action priorities.

To facilitate backend processing, for both hardware and software code generation, the threads are represented in static single-assignment (SSA) form. They interact with the environment of the actor through asynchronous token-based FIFO channels. Their internal communication happens through synchronous unbuffered signals (this is, for instance, how the scheduler triggers actions to fire, and how actions report completion), and they also have shared access to the state variables of the actor.

C. RTL code generation

The final phase of the translation process generates an RTL implementation (in Verilog) from a set of threads in SSA form. The first step simply substitutes operators in expressions for hardware operators, creates the hardware structures required to implement the control flow elements (loops, if-then-else statements), and also generates the appropriate muxing/demuxing logic for variable accesses, including the Φ elements in the SSA form.

The resulting basic circuit is then optimized in a sequence of steps.

(a) **Bit-accurate constant propagation.** This step eliminates constant or redundant bits throughout the circuit, along with all wires transmitting them. Any part of the circuit that does not contribute to the result will also be removed, which roughly corresponds to dead code elimination in traditional software compilation.

(b) **Static scheduling of operators.** By default, operators and control elements interact using a protocol of explicit activation—e.g., a multiplier will get triggered by explicit signals signifying that both its operands are available, and will in turn emit such a signal to downstream operators once it has completed multiplication. In many cases, operators with known execution times can be scheduled statically, thus removing the need for explicit activation and the associated control logic. In case operands arrive with constant time difference, a fixed small number of registers can be inserted into the path of the operand that arrives earlier.

(c) **Memory access optimizations.** Arrays are mapped to Block RAMs (BRAM) for FPGA implementation. These usually small RAM blocks (typically 18 kBits) are distributed across the FPGA, and can be ganged up to form larger memories, or a number of small arrays may be placed into one BRAM. Furthermore, BRAMs usually provide two or more ports, which allows for concurrent accesses to the same memory region. Based on an analysis of the sizes of arrays and the access patterns, the backend maps array variables to Block RAMs, and accesses to specific ports.

(d) **Pipelining, retiming.** In order to achieve a desired clock rate, it may be necessary to add registers to the generated circuit in order to break combinatorial paths, and to give synthesis backends more opportunity for retiming.

	Size slices, BRAM	**Speed** kMB/S	**Code size** kLOC	**Dev. time** MM
CAL	3872, 22	290	4	3
VHDL	4637, 26	180	15	12
Improv. factor	1.2	1.6	3.75	4

Fig. 6. Hardware synthesis results for an MPEG-4 Simple Profile decoder. The numbers are compared with a reference hand written design in VHDL.

Figure 6 shows the quality of result produced by the RTL synthesis engine for a real-world application, in this case an MPEG-4 Simple Profile video decoder. Note that the code generated from the high-level dataflow description actually outperforms the VHDL design in terms of both throughput and silicon area for a FPGA implementation.

VI. Discussion

In this paper it has been shown that CAL, at the same abstraction level, can yield system specifications for direct SW and HW generation. This is made possible by several interesting properties which not only provide appropriate answers at the problem of finding a language that specify systems at high level, but that can also be extremely useful for facing the challenges of next generation digital systems.

Expressing concurrency The intrinsic cability of CAL operators and construct to describe and easily fit concurrent problems make CAL actor-modeling an excellent fit for system design of parallel algorithms and streaming or data dominated applications.

Compactness The MPEG-4 Simple Profile decoder has been manually implemented in different languages such as C/C++, VHDL and CAL. The full implementation is composed of approximately 4000 lines for CAL, 15000 lines for VHDL, 4100 for an optimized version in C/C++. It shows how concise CAL is, but at the same time concurrency and data dependencies are clearly exposed and by raising the level of abstraction of constructs and operators, CAL needs less lines of code to fully describe a given algorithm. Conversely CAL is not overloaded with implementation details and such feature is a clear advantage. CAL focuses only on the description of the algorithm itself and how data is generated and consumed by the different components. Implementation details such as scheduling of the operations are let to code generators.

Analysis CAL language allows the analysis of an actor and networks of actors. For the definition of the actors, CAL provides statically analyzable information about their behavior, such as the number of tokens it produces and consumes in each firing, the necessary conditions for their firing, on what depend those conditions... These information are very useful for effectively schedule, compose, and implement those actors in the final implementations.

Portability CAL specifies algorithms and defines their associated data flow models in a concise and clear way. CAL models are completely independent from final implementation. Thanks to this independency, it eases the integration, exchange and the development of actors. Different implementation for several targets can be built easily from these models. The encapsulation property of the actors is very convenient: global variables do not exist, making the integration of external actors (not written by the designer himself) in the design much easier.

Simplicity of actor design CAL offers a compact, clear and precise semantics, which is tailored to the constraints of actor design and thus facilitates readability and maintainability of the actors. The ease of programming is also necessary to make the language accepted in the scientific community.

Hardware and Software code generation The CAL language is a good starting point for both hardware and software code generation.

The results presented in this paper and in [17] show that efficient hardware code can be generated directly from CAL models. The results show the quality of results produced by the RTL synthesis engine for a real-world application (MPEG-4 Simple Profile video decoder). The code generated from the high-level dataflow description actually outperforms the VHDL design in terms of both throughput and silicon area.

C code can also be generated from CAL models. The results presented in this paper and in [18] show significant improvements compared to CAL dataflow simulation with the Open Dataflow environment [9]. The compilation process has been successfully applied to the same MPEG-4 Simple Profile video decoder. The synthesized software (10600 lines of code) is faster than the CAL dataflow simulated (about 20 frames/s instead of 0.15 frames/s), and close to real-time for

a QCIF format (25 frames/s) on a standard PC platform. It is interesting to notice that the model is scalable: the number of macro-blocks decoded per second remains constant when dealing with larger image sizes.

CAL is particularly well suited for describing signal processing systems which are intrinsically data-driven. It is not by chance that CAL language has been chosen by the ISO/IEC standardization organization in the new MPEG standard called "Reconfigurable Video Coding (RVC)" (ISO/IEC 23001-4 and 23002-4). RVC is a framework allowing users to define a multitude of different codecs, by combining together actors (called coding tools in RVC) from the MPEG standard library that contains video technology from all past standards (i.e. MPEG-2, MPEG-4 etc, etc) . The reader can refer to [19] for more information about RVC. CAL is used to provide the reference software for all coding tools of the entire library. The essential elements of the RVC framework, besides the tool library, include a Decoder Description expressed in an XML dialect which describes in the architecture of the decoder by specifying the connections between the different actors, a Bitstream Schema (BS) which describes the structure, the organization of the data in the bitstream and implicitly defines the parser needed for the specific decoder reconfiguration.

VII. Conclusion and future work

This paper describes a framework based on CAL data flow language that includes a simulator and SW and HW code generators. CAL data flow models results particularly efficient for specifying signal processing systems in a very synthetic form compared to classical imperative languages. Moreover CAL models can be developed to describe architectural features of the desired implementation, thus enabling the designer to work for both algorithm and architecture specification at the same level of abstraction. Hardware and software code generators then provide the implementation of the actors and associated network of actors on different targets (processors or FPGA). CAL succeeds in unifying different levels of abstraction in a single layer at which specification, design space exploration and efficient implementation can be developed. CAL is very expressive and concise. It exposes clearly the intrinsic parallelism of the algorithm by means of the notion of encapsulated actor processing and explicit data dependencies in the actor network. The first promising results obtained by this framework for modelling systems and generating software/hardware implementations, and the recent adoption by ISO/IEC MPEG of CAL as a the new specification formalism for the library that covers in modular form all video algorithms from the different MPEG video coding standards, shows that CAL is an appropriate language for supporting design flows aiming at building complex heterogeneous systems from high level system specifications. Concerning future works and extension of the framework they include the evolution of the software and hardware code generators in terms the extension of the subset of CAL language supported by the two code generators, the development of scheduling tools beyond the SystemC scheduler for mapping on multicore platforms and the evolution of the current Open DataFlow environment with more accurate and extended profiling and debug tools.

References

[1] "Streamc language specification," http://graphics.stanford.edu /streamlang/streamc-3-6-00.pdf.

[2] Alberto Sangiovanni-Vincentelli, "Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design," *Proceedings of the IEEE*, vol. 95, pp. 467–506, 2007.

[3] Petri Kukkala, Mikko Setala, Tero Arpinen, Erno Salminen, Marko Hannikainen, and Timo D. Hamakainen, "Implementing a wlan video terminal using uml and fully automated design flow," *EURASIP J. Embedded Syst.*, vol. 2007, pp. 20–20, 2007.

[4] D. Thomas, "Mda: revenge of the modelers or uml utopia?," *Software, IEEE*, vol. 21, pp. 15–17, 2004.

[5] G. De Micheli, "Hardware synthesis from c/c++ models," in *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, 1999, pp. 382–383.

[6] G. De Micheli, D. Ku, D. Ku, F. Mailhot, and T. Truong, "The olympus synthesis system," *Design & Test of Computers, IEEE*, vol. 7, pp. 37–53, 1990.

[7] Johan Eker and Jörn Janneck, "CAL Language Report," 2003, ERL Technical Memo UCB/ERL M03/48.

[8] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, pp. 127–144, 2003.

[9] "Open dataflow sourceforge project," http://opendf.sourceforge.net/.

[10] Edward A. Lee and Thomas M. Parks, "Dataflow Process Networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[11] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.

[12] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee, "Synthesis of embedded software from synchronous dataflow specifications," *J. VLSI Signal Processing Systems*, vol. 21, no. 2, pp. 151–166, 1999.

[13] Luis Damas and Robin Milner, "Principal type-schemes for functional programs," in *Proceedings of POPL '82*, 1982, pp. 207–212.

[14] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: An Infrastructure for C Program Analysis and Transformation," in *Proceedings of CC 2002*, Apr. 2002, pp. 213–228.

[15] ISO/IEC FDIS 23002-2:2007(E), "Information technology – MPEG video technologies – Part 2: Fixed-point 8x8 inverse discrete cosine transform and discrete cosine transform," 2007.

[16] C. Loeffler, A. Ligtenberg, and G. Moschytz, "Practical Fast 1-D DCT Algorithms with 11 Multiplications," in *Proceedings of ICASSP'89*, Feb. 1989.

[17] Jörn W. Janneck, Ian D. Miller, Dave B. Parlour, Marco Mattavelli, Christophe Lucarz, Matthieu Wipliez, Mickal Raulet, and Ghislain Roquier, "Translating dataflow programs to efficient hardware: an mpeg-4 simple profile decoder case study," in *Design, Automation and Test in Europe (DATE)*, Munich, Germany, 2008.

[18] Matthieu Wipliez, Ghislain Roquier, Mickael Raulet, Jean-Franois Nezan, and Olivier Dforges, "Code generation for the MPEG reconfigurable video coding framework: from CAL actions to C functions," in *IEEE International Conference on Multimedia & Expo (ICME)*, Hannover, Germany, 2008.

[19] Christophe Lucarz, Marco Mattavelli, Joseph Thomas-Kerr, and Jörn Janneck, "Reconfigurable media coding: A new specification model for multimedia coders," in *IEEE Workshop on Signal Processing Systems*, 2007, pp. 481–486.